Methods to Improve Bootup Time in Linux

Tim Bird
Sony Electronics
Bootup Time Working Group Chair
CE Linux Forum

Overview

- Characterization of the problem space
- Current reduction techniques
- Work in progress
- Resources

Characterizing the Problem Space

The Problem

- Linux doesn’t boot very fast
  - Current Linux desktop systems take about 90-120 seconds to boot
- Most CE products must be ready for operation within seconds of boot
  - CELF requirements:
    - boot kernel in 500 milliseconds
    - first available for use in 1 second

Boot Process Overview

1. power on
2. firmware (boot loader) starts
3. kernel decompression starts
4. kernel start
5. user space start
6. RC script start
7. application start
8. first available use

Delay Areas

- Major delay areas in startup:
  - Firmware
  - Kernel/driver initialization
  - User space initialization
    - RC Scripts
    - Application startup

Overview of delays (on a sample desktop system)

Startup Area	Delay
Firmware	15 seconds
Kernel/driver initialization	7 seconds
RC scripts	35 seconds
X initialization	9 seconds
Graphical Environment start	45 seconds
Total:	111 seconds

For laptop with Pentium III at 600 MHZ

Firmware

Firmware/Pre-kernel delays

- X86 firmware (BIOS) is notorious for superfluous delays (memory checking, hardware probing, etc.)
  - Many of these operations are duplicated by the kernel when it starts
- Large delay for spinup of hard drive
- Delay to load and decompress kernel

Reduction Techniques for Firmware

- Custom firmware
  - This is standard in the embedded space anyway
- Don’t load kernel from disk
  - Load kernel from flash
  - Kernel init overlaps with drive spinup
- Kernel Execute-In-Place (XIP)

Typical HD Time-to-Ready

Brand	Size	Time to Ready
Maxtor	3.5”	7.5 seconds
Seagate	3.5”	6.5 - 10 seconds *
Hitachi	3.5”	6 - 10 seconds *
Hitachi	2.5”	4 - 5 seconds
Toshiba	2.5”	4 seconds
Hitachi microdrive	1.0”	1 - 1.5 seconds

* Depends on number of platters

During retries, these times can be extended by tens of seconds, but this is rare.

Load and decompress times

- Typically the kernel is stored in compressed form (zImage or bzImage)
- Entire kernel must be loaded from storage (HD or flash) and decompressed
  - If on HD, there are seek and read latencies
  - If on flash, there are read latencies
- For a slow processor, this can take 1 to 2 seconds

Kernel XIP

- Place kernel uncompressed in flash or ROM
- Map or set kernel text segments directly in machine address space
- Details:
  - Bootloader sets up mapping and transfers control directly to kernel
  - Data segment is copied and bss is initialized, as usual.

Kernel XIP pros and cons

- Pros:
  - Faster bootup – eliminates load and decompress times for kernel
  - Smaller RAM footprint – kernel text segment is not loaded into RAM
- Cons:
  - Adds overhead for running kernel
    - Access to Flash is slower than access to RAM

Kernel XIP results

Boot time for PowerPC 405LP at 266MHZ.

Boot Stage	Non-XIP time	XIP time
Copy Kernel to RAM	85 msec	12 msec *
Decompress kernel	453 msec	0 msec
Kernel time to initialize (time to first userspace prog)	819 msec	882 msecs
Total kernel boot time	1357 msecs	894 msecs

* Data segment must still be copied to RAM

Kernel XIP runtime overhead

Operation	Non-XIP	XIP
stat() syscall	22.4 µsec	25.6 µsec
fork a process	4718 µsec	7106 µsec
context switching for 16 processes and 64k data size	932 µsec	1109 µsec
pipe communication	248 µsec	548 µsec

Results from lmbench benchmark on OMAP (ARM9 168 MHZ)

Kernel

Kernel startup sequence

- start_kernel()
  - initialize architecture
  - init interrupts
  - init memory
  - start idle thread
  - call rest_init()
    - start ‘init’ kernel thread

Driver initializations

- init (kernel thread)
  - call do_basic_setup()
    - call do_initcalls()

      (initialize buses and drivers)
  - prepare and mount root filesystem
  - call run_init_process()
    - call execve("/sbin/init") to start user space process

Measurement Method

- Kernel Function Instrumentation
  - Uses gcc -finstrument-functions option to instrument kernel functions
  - Saves duration of each function call
    - Calls are filtered with a threshold to avoid overload
- See
  - http://tree.celinuxforum.org/pubwiki/moin.cgi/KernelFunctionInstrumentation

Key delays by low-level function

Kernel Function	No. of calls	Avg call time (msec)	Total time (msec)
delay_tsc	5153	1	5537
default_idle	312	1	325
get_cmos_time	1	500	500
psmouse_sendbyte	44	2.4	109
pci_bios_find_device	25	1.7	26
atkbd_sendbyte	7	3.7	26
calibrate_delay	1	24	24

Key delays

- Over 80% of the total bootup time was spent busywaiting in delay_tsc() or spinning in default_idle()
- get_cmos_time() was highly variable (up to 1 second)

High-level delay areas

Kernel Function	Duration in msec
ide_init	3327
time_init	500*
isapnp_init	383
i8042_init	139
prepare_namespace	50*
calibrate_delay	24

** function duration was highly variable*

Delay culprits

- ide_delay_50ms()
  - is called 78 times by children of do_probe()
  - do_probe() is called 31 times by ide_init()
  - results in over 5 seconds of busywaiting
    - (about 70% of total boot time)
- isapnp_init()
- mouse and keyboard drivers
- calibrate_delay()

Reduction Techniques for Kernel

Pre-set loops_per_jiffy

- Very easy to do:
  - Measure once
  - Set value in init/main.c:calibrate_delay_loop()
  - Don’t perform calibration
- Saves about 250 msec (when HZ=100)
- Patch for command line option for this is now in linux-2.6.8-rc1-mm1
  - Use “lpj=xxx” on kernel command line

Use of IDE “noprobe” options

- Can turn off IDE probe with kernel command line:
  - ide<x>=noprobe
  - Requires a bugfix patch (feature was broken in 2.4.20)
- Can also turn off slave devices:
  - eg. hd<x>=none
- Time to probe for an empty second IDE interface was measured at 1.3 seconds

Reduce probing delays

- ide_delay_50ms()
  - accounted for 70% of busywaiting during kernel startup (in test environment)
- Rewrote routine to only delay 5 milliseconds
  - Others have tried as little as 1 millisecond
- Successfully detected and initialized all IDE hardware
- Needs more testing – maybe a config option

Using the “quiet” option

- Overhead of printk output is high
  - serial console speed
  - VGA console
    - slow processor + software scroll = slowness
- Use “quiet” on kernel command line
- Can still read messages from printk buffer after startup (use dmesg)
- 250 msecs savings for 2 different platforms

Avoid RTC Read Synchronization

- get_cmos_time()
  - up to 1 second waiting for RTC clock edge
- Synchronization is not needed in many embedded situations
- Multiple reboot cycles where RTC is set from system time may result in clock drift.
  - Can use other mechanisms to avoid RTC clock error

Use Deferred and Concurrent Driver Initialization

- Change drivers to modules
  - Statically compiled drivers are loaded sequentially, with “big kernel lock” held (in 2.4)
- Replace driver busywaits with yields
- Load drivers later in boot sequence
  - In parallel with other drivers or applications
- Benefit is highly driver-specific
  - e.g. PCI sound card had 1.5 seconds of busywait
- Requires per-driver code changes

Reduction Techniques for User Space

Reduction in RC script overhead

- Use of busybox for shell interpreter (ash) and builtin commands
  - Eliminates overhead of large program invocations
- Modification to RC scripts to avoid loading shell multiple times
- Modification to busybox to avoid fork and exec on shell invocations

Reduction in RC script Overhead Early Results

- Time to run set of RC scripts reduced from 8 seconds to 5 seconds
  - On ARM9, 168 MHZ

Eliminate unneeded RC scripts

Default Script List

anacron.sh
bootmisc.sh
checkfs.sh
checkroot.sh
console-screen.sh
cron.sh
devfsd.sh
devpts.sh
devshm.sh
hostname.sh
hwclock.sh
ifupdown.sh
keymap.sh
modutils.sh
mountall.sh
networking.sh
procps.sh
rmnologin.sh
syslog.sh
urandom.sh

Reduced Script List

bootmisc.sh
checkfs.sh
checkroot.sh
hwclock.sh
modutils.sh
mountall.sh
networking.sh
urandom.sh

Replace RC Scripts with Custom init Program

- Replace scripts and /sbin/init program itself
  - This is already standard procedure for most embedded products
- Use compiled program instead of shell script
  - Avoids shell invocation and parsing overhead
- Drawbacks:
  - You have to maintain your custom init program
  - System is no longer reconfigurable via file operations

Application XIP

- Requires linear file system (like CramFS or ROMFS)
- Map libraries and applications into address space directly from Flash/ROM
- Good application load performance (on first load)
- Slight performance degradation

Application XIP Results

Time to run shell script which starts TinyX X server and *xsetroot -solid red*, then shuts down

Invocation	Non-XIP	XIP
First time	3195 msec	2035 msec
Second time	1744 msec	1765 msec

Defer replay of FS log

- Ext3 and XFS both replay their log at boot/mount time
- Can mount FS readonly on boot
  - Later, switch to read/write and replay the log to ensure FS integrity
- Requires file system areas to be organized to support deferral of access to writable areas.
  - Put writable areas (e.g. /var) in RAM disk temporarily
- About 200 ms improvement in some tests

System-wide improvements

- Reduce kernel, library and application size by using smallest configuration possible.
  - Reduces load time and can improve cache hits
- Keep read-only and executable data separate from writable data in flash storage
  - Write times (which are long) don’t interfere with read times
- Use Linear CramFS for read-only data
  - CramFS has little meta-data and mounts quickly

System-wide improvements (cont.)

- Keep writable files in RAM disk, and migrate to flash after boot
- Reduce the amount of filesystem I/O (especially writes to flash)
- Turn off klogd/syslogd logging to stable storage
- Set library search paths to reduce failed open attempts

Ideas for Future Research

- Driver configuration cache
  - Form of hibernate/unhibernate for device drivers and bus code
- Partial XIP
  - Copy hot-spot portions of kernel into RAM, to overcome performance problem with pure XIP
- Pre-linking
  - Pre-calculate relocations and fixups for dynamic libraries

Instrumentation

- Printk-times
- Kernel Function Instrumentation (KFI) for kernel time measurements
- CELF was working on a timing API proposal
  - We went back to drawing boards based on LKML feedback

Patch Availability

- Some patches are available now.
- Some code is implemented for 2.4, but has not been isolated into patches or ported to 2.6.
  - Code is in CELF 1.0 source tree
- See http://tree.celinuxforum.org/

Primary observation…

Configuration of hardware and software is much more fixed for embedded systems than for desktop systems

Speedup Methods

- Do it faster
- Do it in parallel
- Do it later
- Don’t do it at all

Good Luck!